UCA93 I2C Communications Adapter

Issue 1.0
01/07/2003

Welcome to the Calibre UCA93 adjustable voltage I²C Adapter. This Adapter provides full I²C bi-directional compatibility as either a master or slave from within a Windows 98 2000 or Windows XP environment.

If you have any queries relating to this or any other I²C product supplied by Calibre please visit our web site www.calibreuk.com***.***

For technical support please e-mail techsupport@calibreuk.com ***or send your queries by fax to (44) 1274 730960, for the attention of our I²C Technical Support Department.***

COPYRIGHT

Copyright (c) 2003 Calibre UK Ltd
Cornwall House, Cornwall Terrace
Bradford, BD8 7JS. UK.
E-mail: sales@calibreuk.com
Web site www.calibreuk.com
All World-wide Rights Reserved

Issue 1.0 01/07/2003

Calibre operates a policy of continued product improvement, therefore specifications are subject to change without notice as products are updated or revised.

E&OE.

CALIBRE

Contents

CALIBRE

INTRODUCTION

1.1. General Introduction

The UCA93 is a USB V1.1 to I^2C Adapter based on the PIC 16F874 microcontroller using a Philips PCF8584 for the bus monitor function. It features full I^2C bi-directional compatibility as either a master or slave. I^2C connections are made via a 9 way "D" socket. This product complies with the requirements of EEC Directive 89/336 for EMC and is CE marked.

The UCA93 provides full software control of the I^2C bus voltage, bus speed and pull up resistors. Two spare IO pins are available to the user.

The software functions supplied with the Adapter have been designed to feel as familiar as possible to customers with existing Calibre AT or PCI bus I2C adapters. Nevertheless there are some unavoidable differences between the way the UCA93 works and previous generations of products. Customers are strongly recommended to read Appendix B which deals with migration issues.

New software block functions have been added to take advantage of the USB port's high data rate.

1.2. Packing List

Your I^2C Communications Adapter is supplied with the following items:-

A. I^2C CD ROM
B. The UCA93 Adapter and USB cable

1.3. Configuring the Adapter

NOTE
There are no user adjustable components within the Adapter. The Adapter contains static sensitive devices.

Read the following section to change the configuration.

1.4. Bus Termination and Protection

Normally the system to which the I^2C Communications Adapter is to be connected should already have master pull up resistors fitted to the SCL and SDA lines. If this is not the case, pull up resistors to the +V(adjustable) can be configured as part of the software set up.

1.5. Connecting the Adapter to your System

The USB cable MUST be connected between the PC and the Adapter.

All I^2C connections are made via a 9 way "D" socket:

Pin	Function
1	0V
2	0V
3	0V
4	0V
5	NC
6	SDA (Bi-directional)
7	+V (variable voltage)
8	SCL (Bi-directional)
9	NC

The Adapter also provides two digital input / output pins, which are connected via a 0.1" 4 way PCB connector.

IO 0 is an open drain output which requires an external pull up and can be used on systems which are not 5V tolerant. If IO 0 is used as an input it MUST be connected to a 5V system.

IO 1 is a 5V TTL input and can only be used on 5V tolerant systems.

Pin	Function
1	NC – DO NOT CONNECT
2	IO 0
3	IO 1
4	0V

1.6. Bus Capacitance Limitations/Cable Choice

The maximum allowable capacitance on the I^2C bus in normal mode depends on the value of the SCL and SDA master pull-ups, but never exceeds 400pF. Refer to Phillips Technical Handbook Book 4 Parts 12a and 12b for further details. Care should be taken in choosing a length and type of interconnecting cable, which will not exceed this limit.

For most systems with a distance of a few metres between the I^2C Communications Adapter and the target system, screened cable is NOT recommended, as it is likely to introduce too much capacitance. However, the EMC performance of an unscreened cable is always potentially poorer than a screened one. The Adapter's EMC performance even with an unscreened cable is good - but this may not be true of the target system! If you are in any doubt at to the best way to connect up your system with EMC in mind please contact your supplier or Calibre for advice.

1.7. Variable Voltage Bus Power Supply

Pin 7 on the “D” connector is connected to the variable bus voltage power rail. Power for external circuitry can be drawn from here, but care should be taken never to short it to 0V or to exceed 250mA loading. It is short circuit and overload protected by a self-resetting thermal fuse but prolonged shorting could cause the UCA93 to generate an excessive amount of heat.

The bus voltage is set via software, in the range 2V to 5V.

NOTE that if a bus voltage greater 4.2V is selected the output will be 5V.

CALIBRE

INSTALLING THE ADAPTER UNDER WINDOWS

2.1. Introduction

This section details the installation of the UCA93 I^2C communications Adapter under Windows 98® / Windows2000® and Windows XP®.

The appearance of the dialog boxes during the installation of new hardware varies depending on the version of Windows.

The Adapter supports plug and play via the USB port.

2.2. Installing the Adapter

Connect the Adapter to the PC using the USB cable, the power light will illuminate.

The operating system will detect the installation of new hardware, follow the set up wizard instructions.

Either browse the CD or select the “have disc” option select the FTD2XX.inf.

The wizard will then install the device drivers.

IMPORTANT NOTE

Windows XP includes a driver for the USB interface device used in the Adapter, and will normally install this. The driver installed from the XP disc **WILL NOT WORK** with this Adapter.

If you are using Windows XP you can check that the correct driver is installed by

1) Right click on the “My Computer Icon”
2) Select properties
3) Select the Hardware Tab and click “Device Manager”
4) Expand the “Universal Serial Bus Controllers”
5) If the driver is provided by Microsoft you need to select the update driver then browse for and select the FTD2XX.inf
6) Once installed correctly the FTDI FT8U2XX Device should be listed

PLEASE NOTE The FTD2XX.inf is NOT digitally signed but is correct.

3.1. Introduction

Each utility is documented in a standard format which lists its name, usage, function and effect on the Adapter is given. The Adapter should be setup prior to any data transfer.

Within the DLL there are two versions of some functions e.g. SendAddress and SendAddressNoStatus, the first provides legacy support for software written for other Calibre UK Ltd I2C products, the second format does not return the status.

3.1.1. Legacy Function

The following changes have had to be made to the legacy functions

1) The parameters passed to the setup function are specific to the Adapter
2) The status wait has been removed and is now controlled by hardware, the Adapter will wait up to 500 micro seconds before returning an error code indicating that the transfer failed.
3) The Setnack is now changed on the read you require the acknowledge state to change NOT on the byte before.
4) There is no longer a need to perform a Trash read after a restart.

3.1.2. No Status Functions

The xxNoStatus functions provide faster transfers as they do not require the USB bus to change direction between writing and reading. The function prototypes are included in the manual but care MUST be taken when using these functions not to corrupt the I^2C with uncontrolled transfers. These functions are ONLY to be used when users are 100% certain that all devices on the I^2C WILL acknowledge their address and that there is a single master.

PLEASE NOTE NO TECHNICAL SUPPORT IS AVAILABLE FOR THESE FUNCTIONS

Calibre UK recommend using the block functions detailed below

3.1.3. Block Functions

To aid users wishing to transfer data without the complication of having to write a I^2C transfer protocol a number of Block functions have been added. DO NOT exceed the data transfer limitations described in the individual functions as this will cause errors and data corruption.

C and C++ users will undoubtedly wonder why they are passing data to the DLL when they could much more efficiently pass a pointer, unfortunately not all the development environments this Adapter will be used on support pointers.

3.2. Function Prototypes

If you are using 'C' or 'C++' copy the file CALUCA.H into the directory containing your project and add the line:

```
#include " CALUCA.H"
```

The following functions are implemented in the windows libraries:-

```
extern __declspec(dllimport) int WINAPI Setup (int, int, int, int, int);
extern __declspec(dllimport) int WINAPI SendAddress (int, int);
extern __declspec(dllimport) int WINAPI WriteByte(int);
extern __declspec(dllimport) int WINAPI ReadByte(int);
extern __declspec(dllimport) int WINAPI SendStop(void);
extern __declspec(dllimport) int WINAPI Restart (int, int);
extern __declspec(dllimport) int WINAPI GetStatus(void);
extern __declspec(dllimport) int WINAPI recover(void);
extern __declspec(dllimport) void WINAPI SlaveLastByte(void);
extern __declspec(dllimport) int WINAPI DLLVersion(void);


extern __declspec(dllimport) void WINAPI SendAddressNoStatus(int , int, int);
extern __declspec(dllimport) void WINAPI RestartNoStatus(int , int, int );
extern __declspec(dllimport) void WINAPI WriteByteNoStatus(int );
extern __declspec(dllimport) void WINAPI ReadByteNoStatus(int );
extern __declspec(dllimport) void WINAPI SendStopNoStatus(void);


extern __declspec(dllimport) int WINAPI SetBlockData(int);
extern __declspec(dllimport) int WINAPI BlockWriteStatus (void);
extern __declspec(dllimport) int WINAPI BlockWrite(int , int , int ,int, int );
extern __declspec(dllimport) int WINAPI GetBlockData(void);
extern __declspec(dllimport) int WINAPI BlockReadStatus(void);
extern __declspec(dllimport) int WINAPI BlockRead(int , int , int ,  int, int );
extern __declspec(dllimport) int WINAPI BlockSlaveTransmitterStatus(void);
extern __declspec(dllimport) int WINAPI BlockSlaveTransmitter( int );
extern __declspec(dllimport) int WINAPI BlockSlaveReceiverStatus(void);
extern __declspec(dllimport) int WINAPI BlockSlaveReceiver(int , int );
extern __declspec(dllimport) int WINAPI Write_IO_Pin (int , int );
extern __declspec(dllimport) int WINAPI Read_IO_Pin (int );
extern __declspec(dllimport) int WINAPI CheckBusVoltage(void);
extern __declspec(dllimport) int WINAPI CheckPullupVoltage(void);
```

If you are using Visual Basic copy the file UCA93LV.BAS into the directory containing your project and add the file UCA93LV.BAS to your project:

The following functions are implemented in the windows libraries:-

```
Public Declare Function Setup% Lib "USBDLL_XP.dll" (ByVal OwnAddress%, ByVal Sclk%, ByVal
BusVoltage%, ByVal PullUpsOn%, ByVal SlaveBlockTimeout%)
Public Declare Function SendAddress% Lib "USBDLL_XP.dll" (ByVal SlaveAddress%, ByVal setnack%)
Public Declare Function Restart% Lib "USBDLL_XP.dll" (ByVal SlaveAddress%, ByVal setnack%)
Public Declare Function WriteByte% Lib "USBDLL_XP" (ByVal wrdata%)
Public Declare Function ReadByte% Lib "USBDLL_XP.dll" (ByVal setnack%)
Public Declare Function SendStop% Lib "USBDLL_XP.dll" ()
Public Declare Function GetStatus% Lib "USBDLL_XP.dll" ()
Public Declare Function Recover% Lib "USBDLL_XP.dll" ()
Public Declare Function SlaveLastByte% Lib "USBDLL_XP.dll" ()
Public Declare Function DLLVersion% Lib "USBDLL_XP.dll" ()
Public Declare Function SetBlockData% Lib "USBDLL_XP.dll" (ByVal DataVal%)
Public Declare Function GetBlockData% Lib "USBDLL_XP.dll" ()

Public Declare Function BlockSlaveTransmitter% Lib "USBDLL_XP.dll" (ByVal TimeOut%)
Public Declare Function BlockSlaveTransmitterStatus% Lib "USBDLL_XP.dll" ()
Public Declare Function BlockSlaveReceiver% Lib "USBDLL_XP.dll" (ByVal NoBytesToTransmit%,
ByVal TimeOut%)
Public Declare Function BlockSlaveReceiverStatus% Lib "USBDLL_XP.dll" ()
```

NOTE A type is defined in UCA93LV.BAS to help passing parameters to the DLL, if you do not wish to use this local variables MUST be declared as static

3.3. Function Description

3.3.1. Setup

Function specification: int Setup(int OwnAddress, int ClockSpeed, int BusVoltage, int PullUpsOn, int SlaveBlockTimeout)

Parameters are:

int ownaddress

This is the I2C address to which the Adapter is to respond in slave mode. This forms the upper 7 bits of the 8 bit address, the lowest bit being the read(1) or write(0) bit. This means that if ownaddress = 57H the card will respond to a write address of AEH and a read address of AFH.

int ClockSpeed

This is the clock rate (bit rate for the I^2C serial bus) when operating as a master. For 400kHz enter 400, for 90kHz enter 90.
Maximum value 400 minimum value 25

int BusVoltage

The desired bus voltage multiplied by 100 e.g. 330 would give a bus voltage 3.3V. Please note that if a value between 4.2V and 5V is requested the output will be 5V.
Maximum value 500 (5V) Minimum value 180 (1.8V).

int PullUpsOn

Set (1) to turn the pull up resistors on, clear (0) to turn them off.

int SlaveBlockTimeout

This parameter is only used by the slave block functions. If the Master has not completed the transfer in this period (seconds) the Adapter will abort the transfer and return an error code.
Range 1 – 255 seconds. NOTE: setting this value to 0 will prevent the Adapter from timing out the slave block function.

Parameters returned: If the software fails to find the driver error code 8000H is returned otherwise the status is returned.

Prerequisites: None.

Functional description: This function initialises Adapter ready for I^2C transfers.

3.3.2. SendAddress

Function specification: Int SendAddress(int SlaveAddress, int SetNack)

Parameters are:

int SlaveAddress

This is the address to be accessed via the I2C, e.g. A0H.

int SetNack

This controls whether the Adapter transmits an acknowledge down the I^2C bus on reception of a byte. The last byte received during a transfer must not be acknowledged, in all other cases acknowledge must be enabled. If SetNack = 0 then acknowledge is enabled, if SetNack = 1 then acknowledge is disabled. Normally there will be another transfer (data read or write) after the SendAddress and so SetNack is enabled (0);

Parameters returned: ***int ErrCode***.

If the transfer time out occurs error code 8001H is returned otherwise the status is returned.

Prerequisites: The Adapter must be configured by running **Setup**.

Functional description: The function waits for the bus to be free then sends the slave address.
The function waits for the address to be sent. Should a time-out occur during the sending of an address then an error code 8001H is returned, otherwise the status is returned.

WriteByte

Function specification	Int WriteByte(int DataByte)
Parameters are:	***int DataByte***
	This is the byte of data to be written.
Parameters returned	***int ErrCode***.
	If the transfer time out occurs error code 8002H is returned otherwise the status is returned.
Prerequisites	Adapter must be configured using **Setup**, start and write address sent by **SendAddress**.
Functional description	The function writes the data to the Adapter and then waits for it to be sent. Should a time-out occur during the sending of the data then error code 8002H is returned, otherwise the status is returned.
	WriteByte is compatible with both master write and slave write modes.

3.3.3. ReadByte

Function specification	Int ReadByte(int SetNack)
Parameters are:	***int SetNack***
	This controls whether the Adapter transmits an acknowledge down the I^2C bus on reception of a byte. The last byte received during a transfer must not be acknowledged, in all other cases acknowledge must be enabled. If SetNack = 0 then acknowledge is enabled, if SetNack = 1 then acknowledge is disabled.
Parameters returned	***int I2CData***
	If a time-out occurs the ErrCode 8003H is returned, otherwise the data is returned.
Prerequisites	Adapter must be configured using **Setup**, start and read address sent by **SendAddress**.
Functional description	The data is read from the Adapter.
	ReadByte is compatible with both master read and slave read modes.

3.3.4. SendStop

Function specification	Int SendStop(void)
Parameters are:	***None***
Parameters returned	***int ErrCode***.
	If the transfer time out occurs error code 8006H is returned otherwise the status is returned.
Prerequisites	Adapter must be configured using **Setup.** Should normally only be used at the end of a transmission. Correct acknowledge sequence must have been applied if the transmission was a read.
Functional description	Instructs the Adapter to send a stop code and wait for it to be sent.
	Should a time-out occur during the sending of a stop then an error code 8006H is returned, otherwise the status is returned.

3.3.5. Restart

Function specification	Int Restart(int SlaveAddress, int SetNack)
Parameters are:	***int slaveaddress***
	The address to be accessed via the I2C, e.g. A1H.
	int SetNack
	This controls whether the Adapter transmits an acknowledge down the I^2C bus on reception of a byte. The last byte received during a transfer must not be acknowledged, in all other cases acknowledge must be enabled. If SetNack = 0 then acknowledge is enabled, if SetNack = 1 then acknowledge is disabled. It is normal to read data after a restart and hence SetNack will normally be 0.
Parameters returned	***int ErrCode***
	If the transfer time out occurs error code 8004H is returned otherwise the status is returned.
Prerequisites	Adapter must be configured using **Setup.** A start and slave address must have previously been sent using **SendAddress**.

	Usually a data pointer would already have been written using **WriteByte.**
Functional description	Sends a start code and the slave address. The function waits for the address to be sent. Should a time-out occur during the sending of an address then an error code 8004H is returned, otherwise the status is returned.

3.3.6. GetStatus

Function specification*:*	Int GetStatus(void)
Parameters are:	None
Parameters returned	***int I2Cstatus*** The current value of the bus status is returned.
Prerequisites	Adapter must be configured using **Setup.**
Functional description	The function reads status word from the Adapter and returns it.

3.3.7. Recover

Function specification	Int Recover(void)
Parameters are:	None
Parameters returned	***int ErrCode.*** If the bus recovery failed error code 800FH is returned otherwise the status is returned.
Prerequisites	Adapter must be configured using **Setup.**
Functional description	This function clears all USB buffers and issues three consecutive stop commands on the bus, with a delay of about 5uS in between. It then clears the Adapter registers and reads the status. This should normally set the Adapter into a known idle state when a bus error or other problem has occurred. If the status does not indicate bus free or the Bus Error bit is still set then 800FH is returned otherwise the status is returned.

3.3.8. SlaveLastByte

Function specification	void SlaveLastByte(void)
Parameters are:	***None***
Parameters returned	***None***.
Prerequisites	Adapter must be configured using **setup.** This function would normally only be called following the end of a transmission in slave write mode - when the Adapter is being read as a slave, by another master, *not when writing to a slave using the Adapter.*
Functional description	This function is used when the Adapter is a slave being read by a master elsewhere on the bus - the Adapter is in slave write mode. The function must be called immediately after the master indicates the last byte has been read (by not acknowledging that byte). This function is required to clear the I^2C data lines so that the master can send a stop signal.

3.3.9. BlockWrite

Function specification	void BlockWrite(int SlaveWriteAddress, int MSB_WordAddress, int LSB_WordAddress,int NoBytesToSend, int NoTries)
Parameters are:	***int SlaveAddress*** This is the address to be accessed via the I2C, e.g. A0H ***int*** MSB_WordAddress, int LSB_WordAddress This is the location within the slave for the data. The use of two bytes allows users to communicate with devices requiring two byte pointers e.g. 24C64 EEPROMS. If one (or both) of the word addresses are not required set the value to a number greater than 255 and it will not be transmitted. ***int NoBytesToSend*** The number of bytes to be sent which MUST be pre loaded into the driver buffers. The maximum size of this buffer is 2048 exceeding this **WILL** cause the data to be corrupted. ***Int NoTries*** This is the number of times the Adapter is to try to send the address.
Parameters returned	None.
Prerequisites	Adapter must be configured using **Setup.** The data MUST be loaded into the driver buffers PRIOR to calling this functions (see SetBlockData).
Functional description	This function causes the Adapter to send a block of data via the I2C bus.

3.3.10. SetBlockData

Function specification	*int* SetBlockData(int DataVal)
Parameters are:	***Int DataVal***
Parameters returned	***Returns an unused integer***.
Prerequisites	Adapter must be configured using **setup**.
Functional description	This stores data in the driver buffers read to be sent by the BlockWrite function. MAXIMUM DATA SIZE 2048 bytes.

3.3.11. BlockWriteStatus

Function specification	int BlockWriteStatus (void))
Parameters are:	***None***
Parameters returned	***int TransferStatus***. Returns 0 if the transfer has not been completed. Returns the I2C Bus status if the transfer is completed. Returns 9009H if the transfer time out.
Prerequisites	Adapter must be configured using **Setup**. Data must have been sent using the BlockWrite function.
Functional description	Returns the current status of the BlockWrite transfer NOTE this function will time out if the block is not sent in NoBytes / 100 seconds (or 1 second which ever is the greater).

3.3.12. BlockRead

Function specification	void BlockRead(int SlaveAddress, int MSB_WordAddress, int LSB_WordAddress, int NoBytesToRead, int NoTries)
Parameters are:	***int SlaveAddress*** This is the address to be accessed via the I2C, e.g. A0H If the SlaveAddress is is a read address then the Word Addresses WILL NOT be sent. ***int*** MSB_WordAddress, int LSB_WordAddress This is the location within the slave for the data. The use of two bytes allows users to communicate with devices requiring two byte pointers e.g. 24C64 EEPROMS. If one (or both) of the word addresses are not required set the value to a number greater than 255 and it will not be transmitted. ***int NoBytesToRead*** The number of bytes to be sent which MUST be pre loaded into the driver buffers. The maximum size of this buffer is 2048 exceeding this **WILL** cause the data to be corrupted. ***Int NoTries***

	This is the number of times the Adapter is to try to send the address.
Parameters returned	None.
Prerequisites	Adapter must be configured using **setup.**
Functional description	This function causes the Adapter to read a block of data via the I2C bus.

3.3.13. GetBlockData

Function specification	*int* GetBlockData(void)
Parameters are:	***None***
Parameters returned	***int RdData***.
Prerequisites	Adapter must be configured using **Setup**. The block read function MUST have been called and the BlockReadStatus should be called to determine if the transfer has been completed
Functional description	Read data from the driver buffers. MAXIMUM DATA SIZE 2048 bytes.

3.3.14. BlockReadStatus

Function specification	int BlockReadStatus (void))
Parameters are:	***None***
Parameters returned	***int TransferStatus***. Returns 0 if the transfer has not been completed. Returns the I2C Bus status if the transfer is completed. Returns 900AH if the transfer time out.
Prerequisites	Adapter must be configured using **Setup**. The transfer must be started using the BlockRead function.
Functional description	Returns the current status of the BlockWrite transfer NOTE this function will time out if the block is not sent in NoBytes / 100 seconds (or 1 second which ever is the greater).

3.3.15. BlockSlaveTransmittter

Function specification	int BlockSlaveTransmitter(int NoBytesToTransmit, int Timeout)
Parameters are:	***int NoBytesToTransmit*** The number of bytes to be sent which MUST be pre loaded into the driver buffers. The maximum size of this buffer is 2048 exceeding this **WILL** cause the data to be corrupted. ***Int Timeout*** If the transfer is not completed within Timeout seconds the BlockSlaveTansmitStatus will return a Timeout error.
Parameters returned	***Returns an unused integer***.
Prerequisites	Adapter must be configured using **Setup.** The data MUST be loaded into the driver buffers PRIOR to calling this functions (see SetBlockData).
Functional description	This function causes the Adapter to respond to I2C master requests for data. This function supports transfers where the master sends WordAddress pointers to data locations within the transfer block. If the master does not request to read data from the beginning of the data block the Adapter will remove the data from the buffer until the byte requested by the master is reached it will then transmit the data. Should the master exceed the number of bytes available the Adapter will honour all requests for data and the overflow will be reported see BlockSlaveTransmitterStatus. This function allows the user to emulate simple slave devices e.g. EEPROM BUT once the master has completed the transfer the Adapter will NOT respond to any further requests for data until the BlockSlaveTransmitter is called again.

3.3.16. BlockSlaveTransmitterStatus

Function specification	int BlockSlaveTransmitterStatus (void))
Parameters are:	***None***
Parameters returned	***int TransferStatus***. Returns 0 if the transfer has not been completed. Returns the a status if the transfer is completed (see below). Bit 0 Set if a write address received Bit 1 Set if word address 0 received Bit 2 Set if word address 1 received Bit 3 Set if restart received Bit 4 Stop detected Bit 5 Set if the transfer timeout (Value in Setup function) occurred Returns 900BH if the transfer time out.
Prerequisites	Adapter must be configured using **Setup**. The transfer must be started using the BlockSlaveTransmittter function.
Functional description	Returns the current status of the BlockSlaveTransmittter transfer NOTE this function will time out the BlockSlaveTransmittter function. Read two bytes of data from the Adapter using GetBlockData to determine where the master read from in the block (high byte first), read the next two bytes to determine how many bytes of data the master read from the slave (high byte first), read 1 byte to determine the status.

3.3.17. BlockSlaveReceiver

Function specification	int BlockSlaveReceiver (int NoBytesToReceive, int Timeout)
Parameters are:	***int NoBytesToReceive*** The number of bytes to expected from the master. NOTE if the protocol requires the master to transmit one or more word address bytes these MUST be included in the NoBytesToReceive. The maximum size of this buffer is 2048 exceeding this **WILL** cause the data to be corrupted. **PLEASE NOTE THIS MUST INCLUDE THE NUMBER OF WORD ADDRESS BYTES IF THEY ARE TRANSMITTED BY THE MASTER** ***Int Timeout*** If the transfer is not completed within Timeout seconds the BlockSlaveReceiverStatus will return a Timeout error.
Parameters returned	***Returns an unused integer***.
Prerequisites	Adapter must be configured using **Setup.**
Functional description	This function causes the Adapter to respond to I2C master transmission of data. This function supports transfers where the master sends WordAddress pointers to data locations within the transfer block. Should the master exceed the number of bytes expected the Adapter will honour all transfers for data and the overflow will be reported see BlockSlaveReceiverStatus. This function allows the user to emulate simple slave devices e.g. EEPROM BUT once the master has completed the transfer the Adapter will NOT respond to any further requests for data until the BlockSlaveReceiver is called again.

3.3.18. BlockSlaveReceiverStatus

Function specification	int BlockSlaveReceiverStatus (void))
Parameters are:	***None***
Parameters returned	***int TransferStatus***. Returns 0 if the no data has been received. Returns the a status if the transfer is completed (see below). Bit 0 Not Used Bit 1 Not Used Bit 2 Not Used Bit 3 Not Used Bit 4 Stop detected Bit 5 Set if the transfer timeout (Value in Setup function) occurred Returns 900DH if the expected number of bytes is exceeded Returns 900CH if the transfer time out.
Prerequisites	Adapter must be configured using **Setup**. The transfer must be started using the BlockSlaveReceiver function.
Functional description	Returns the current status of the BlockSlaveReceiver transfer NOTE this function will time out the BlockSlaveReceiver function. Read two bytes of data from the Adapter using GetBlockData to determine how many bytes of data the master read from the slave. Read the first two bytes to determine how many bytes of data the master read from the slave (high byte first), read 1 byte to determine the status.

3.3.19. CheckBusVoltage

Function specification	int CheckBusVoltage (void))
Parameters are:	***None***
Parameters returned	***int BusVoltage*** Returns the I2C bus output voltage multiplied by 100. Returns 0x800C if the transfer timed out.
Prerequisites	Adapter should be configured using **Setup**. If this is not done the I2C bus voltage will default to 3.3V.
Functional description	Returns the current bus voltage multiplied by 100 as measured on pin 7 of the output socket. For example, will return 330 for a 3.3V bus voltage , 500 for a 5.0V bus voltage and 200 for a 2.0V bus voltage.

3.3.20. CheckPullupVoltage

Function specification	int CheckPullupVoltage (void))
Parameters are:	***None***
Parameters returned	***int PullupVoltage*** Returns the I2C pullup voltage multiplied by 100 measured upstream of the output overcurrent protection device. Returns 0x800C if the transfer timed out.
Prerequisites	Adapter should be configured using **Setup**. If this is not done the I2C bus voltage will default to 3.3V.
Functional description	The pullup voltage is the same as the I2C output voltage but measured upstream of the resettable polyfuse protection device. Returns the current pullup voltage multiplied by 100. For example, will return 330 for a 3.3V pullup voltage , 500 for a 5.0V pullup voltage and 200 for a 2.0V pullup voltage. If the pullup voltage differs from the bus voltage (measured by the previous function) by more than a few tens of millivolts then too much current is being drawn from the Adapter. No more than 250mA should be drawn from the Adapter.

3.3.21. Write_IO_Pin

Function specification int Write_IO_Pin (int IONumber, int IOState)
Parameters are: ***int IONumber***
The number of the IO pin (0 or 1) to be driven
int IOState
The required state (0 or 1) of the selected IO pin
Parameters returned ***int***
Returns a 0 if function successful.
Returns 0x800D if the transfer timed out.
Prerequisites ***None***
Functional description Two spare IO pins are available to drive external logic. IO0 is on PL1 pin 2 and is an open drain driver. When driven low it can sink up to 10mA, when high it is high impedance and cannot source current. A pullup resistor should be fitted to the desired rail which should not exceed +5V. IO1 is on PL1 pin 3 and is a complementary driver which can source or sink up to 10mA. Its logic levels are fixed at 0V and +5V nominal.

3.3.22. Read_IO_Pin

Function specification int Read_IO_Pin (int IONumber)
Parameters are: ***int IONumber***
The number of the IO pin (0 or 1) to be read
Parameters returned ***int***
Returns the state (0 or 1) of the chosen IO pin if function successful.
Returns 0x800D if the transfer timed out.
Prerequisites ***None***
Functional description Two spare IO pins are available which can be driven by external logic. IO0 is on PL1 pin 2 and IO1 is on PL1 pin 3. These pins are normally high impedance input unless specifically driven by the Write_IO_Pin function. They can be driven high or low by external logic then read by the Read_IO_Pin function. Normal 5V CMOS logic levels (0 < 1.5V, 1 > 3.5V) should be applied for reliable reading.

3.3.23. SendAddressNoStatus

Function specification	void SendAddressNoStatus(int SlaveAddress, int SetNack, int NoTries)
Parameters are:	***int SlaveAddress*** This is the address to be accessed via the I2C, e.g. A0H. ***int SetNack*** This controls whether the Adapter transmits an acknowledge down the I2C bus on reception of a byte. The last byte received during a transfer must not be acknowledged, in all other cases acknowledge must be enabled. If SetNack = 0 then acknowledge is enabled, if SetNack = 1 then acknowledge is disabled. Normally there will be another transfer (data read or write) after the SendAddress and so SetNack should be enabled (0); ***Int NoTries*** This is the number of times the Adapter is to try to send the address.
Parameters returned	***None***.
Prerequisites	The Adapter must be configured by running **Setup**.
Functional description	The function waits for the bus to be free. Then sends the slave address with the appropriate acknowledge. The acknowledge must equal 0. The function will generate a start and send the address. If the slave does not acknowledge then the Adapter will generate a stop. It will try to send the address a number of times equal to NoTries.

3.3.24. WriteByteNoStatus

Function specification	void WriteByteNoStatus(int DataByte)
Parameters are:	***int DataByte*** This is the byte of data to be written.
Parameters returned	***None***.
Prerequisites	Adapter must be configured using **Setup**, start and write address sent by **SendAddressNoStatus**.
Functional description	The function writes the data to the Adapter and then waits for it to be sent. WriteByteNoStatus is compatible with both master write and slave write modes.

3.3.25. ReadByteNoStatus

Function specification	Int ReadByte(int SetNack)
Parameters are:	***int SetNack*** This controls whether the Adapter transmits an acknowledge down the I2C bus on reception of a byte. The last byte received during a transfer must not be acknowledged, in all other cases acknowledge must be enabled. If SetNack = 0 then acknowledge is enabled, if SetNack = 1 then acknowledge is disabled.
Parameters returned	***None***
Prerequisites	Adapter must be configured using **Setup**, start and read address sent by **SendAddressNoStatus**.
Functional description	The data is read from the Adapter. ReadByteNoStatus is compatible with both master read and slave read modes.

3.3.26. SendStopNoStatus

Function specification	void SendStopNoStatus(void)
Parameters are:	***None***
Parameters returned	***None***.
Prerequisites	Adapter must be configured using **Setup.** Should normally only be used at the end of a transmission.
Functional description	Instructs the Adapter to send a stop code and wait for it to be sent.

3.3.27. RestartNoStatus

Function specification	void RestartNoStatus(int SlaveAddress, int SetNack, int NoTries)
Parameters are:	***int SlaveAddress*** The address to be accessed via the I2C, e.g. A1H. ***int SetNack*** This controls whether the Adapter transmits an acknowledge down the I^2C bus on reception of a byte. The last byte received during a transfer must not be acknowledged, in all other cases acknowledge must be enabled. If SetNack = 0 then acknowledge is enabled, if SetNack = 1 then acknowledge is disabled. It is normal to read data after a restart and hence SetNack will normally be 0. ***Int NoTries*** This is the number of times the Adapter is to try to send the address
Parameters returned	***None***
Prerequisites	Adapter must be configured using **Setup.** A start and slave address must have previously been sent using **SendAddressNoStatus**. Usually a data pointer would already have been written using **WriteByteNoStatus.**
Functional description	Sends a start code and the slave address.

The Real-Time Bus Monitor

To install the monitor run the setup program located in the \CD_USB\USB_Monitor folder, follow the instructions given by the installation wizard.

Before attempting to run the monitor program ensure that the device drivers are installed correctly in accordance with this manual.

The program UI2C_MONITOR.EXE is a windows based non-invasive real-time bus monitor which records activity on an I^2C-bus. The transfers are processed into a simple to read format which can be saved to the PC hard disc.

The monitor can be configured to capture all transfers or alternatively it can capture all transfers between a start and end address. These features allow the user to capture only those transfers which are of interest.

Every effort has been made to ensure that the monitor captures all of the transfers it is possible that the program may miss part of a transfer on heavily loaded I2C buses, this is due to the operating system servicing other hardware and / or applications. To minimise the amount of data missed we recommend that you close as many windows applications as possible especially all windows messaging applications, e-mail and FindFast. We also recommend that you run the application from a 900MHz PC (or better), note the application will run on a lesser specification PC but you are likely to miss more data.

The following mnemonics are used by the monitor program.

SaXX Start code followed by address, acknowledged.

SnXX Start code followed by address, not acknowledged.

DaXX Data byte, acknowledged.

DnXX Data byte, not acknowledged.

STOP Stop code.

BUS ERROR A bus error has occurred. This is non-fatal to the monitor but a small amount of data may have been lost whilst recovery was taking place.

Appendix A I²C Communications Adapter Status Codes

This is an eight bit register, read using the GetStatus routine. Each individual bit has its own meaning as follows:

Bit 7 (MSB) - The (old) PIN Bit

To all intents and purposes, this bit is now redundant. Previous Calibre I2C products used this bit to synchronise data transfers between the I2C bus and the host PC. On the UCA93 synchronisation is achieved by the Adapter returning a status byte to the host when the requested function has completed or timed out. Whereas the value of the PIN bit used to be important, now it is whether or not the status byte has been returned to the host that is important. The PIN bit roughly mimics the behaviour of the old products so that customers still feel comfortable with it, but software should not rely on it. Other bits in the status word are still important, and should be interpreted as below.

Bit 6 – The Timeout Bit Normally this bit should read as a 0. If it is set the Adapter timed out when attempting some master activity. This could mean that another master has control of the bus or that a bus error has occurred or that a slave device is seriously slowing down the bus. Timeout normally occurs about 500uS after a master function is requested. This is a new bit which was not implemented on previous generations of Calibre products.

Bit 5 - The (old) STS Bit

This bit normally reads as a 0 and is set this way only as legacy support. On previous Calibre I2C products indicated that a stop had been detected whilst in slave receiver mode. Software should not rely on this bit.

Bit 4 - The BER (Bus Error) Bit

This bit should normally read as a 0. It is set high when a misplaced Start or Stop has been detected. This can be quite serious since the I²C devices on the bus may be left in an undefined state after a bus error has occurred - in some circumstances the only way to get the bus going again may be to reset all the I²C devices on it.

Bit 3 - The Ack Bit

This bit indicates the state of the 9th I2C bit which is the acknowledge bit. It is active low so reads as 0 for if the receiver acknowledged and 1 if the receiver did not acknowledge. In a master SendAddress or Restart function it indicates whether the slave is present or not. In a master write operation it indicates whether the slave acknowledged the data byte or not, in a master read it indicates whether or not the master was requested to acknowledge the data from the slave or not.

Bit 2 - The (old) AAS Bit

This bit should normally read as a 0 and is set this way for legacy support. On previous Calibre I2C products indicated that a device had been addressed whilst in a slave mode. Software should not rely on this bit.

Bit 1 - The LAB (Lost Arbitration) Bit

This bit should read as a 0. It is set = 1 when, in multimaster operation (more than one master present on the I²C bus) arbitration is lost to another master on the I²C bus.

Bit 0 - The BB (Bus Busy) Bit

This bit reads as a 1 when the bus is idle. Once a Start has been detected it goes low and stays low until the transfer is terminated correctly with a Stop. New transmissions should therefore not be attempted if this bit is low – it may be that another master has control. If there is no other master on the system it may be that a slave is stuck or a bus error has occurred in which case a Recover may free up the bus.

Appendix B Migration Notes for Existing Calibre I2C Customers

These notes are intended for customers who have code working on one or more of Calibre UK's existing AT or PCI based adapters and wish to convert it to work with the new UCA93 Adapter. The fundamental differences are a much reduced reliance on polling the status register, a new way of calling slave functions and the introduction of new block transfer functions.

USB Bus Characteristics

The USB V1.1 bus is fast – data transfer rates can approach a 1 million 8-bit bytes per second. A brief period of USB bus housekeeping is performed at 1mS intervals then the rest of that 1mS time slice is available for data transfer to a particular peripheral in a particular direction. All transfers are initiated by the host PC – it is very much in charge of the system. Direction changes cost bandwidth because even a single byte transfer in a particular direction will occupy a time slice of 1mS. Single byte transfers in alternate directions will effectively reduce the bus bandwidth to 1kHz! – similar to RS232. Software functions for the UCA93 are fundamentally designed to reduce data direction changes and that is why some of the traditional Calibre software method have had to be re-thought.

Polling the Status Register (and how to avoid it)

Traditional Calibre software requests a function (say SendAddress) then polls the status register until the status indicates that the function has completed. On the UCA93 this is replaced by a function of the type

I2CStatus = SendAddress(SlaveAddress);

which instructs the adapter to generate the start transmit the address then waits for a single status byte to be returned from the adapter indicating how the transfer went. See Appendix A for details. One status byte will always be returned even if the function fails and times out (about 500uS). Multiple polls are not needed nor desirable and should be removed from code.

Previous Calibre I2C products used the PIN bit to synchronise data transfers between the I2C bus and the host PC but this now is all handled by the PIC16F874 microcontroller on the Adapter. Communication between the Adapter and host is now synchronised differently so to all intents and purposes this bit is now redundant.

Block Functions

Block functions are I2C master functions designed to take advantage of the high bandwidth of the USB bus whilst minimising data direction changes which slow the bus down. Instead of piecing together I2C functions the user can for example load data into an array, then call a BlockWrite function.

Block Write Function

On receiving the BlockWrite function the adapter will generate the start and address, then optionally a single or double pointer followed by a block of data up to 2048 bytes which is preset by the SetBlockData function. If the slave device does not acknowledge, the adapter will send a stop after the address, then repeat the sequence again up to Tries times to rouse the slave. If the slave still does not acknowledge, the function will terminate cleanly. The user should keep track of the BlockWrite function by immediately calling the BlockWriteStatus function which will return the status byte indicating how the BlockWrite function got on and how it terminated.

Block Read Function

On receiving the BlockRead function the adapter will generate the start and address, then optionally a single or double pointer. If the slave device does not acknowledge, the adapter will send a stop after the address, then repeat the sequence again up to Tries times to rouse the slave. If the slave still does not acknowledge, the function will terminate cleanly. The user should keep track of the BlockRead function by immediately calling the BlockReadStatus function which will return the status byte indicating how the BlockRead function got on and how it terminated. The data returned from the slave to the host via the adapter can be accessed by calling the BlockRead function which accesses a global variable.

Slave Functions

The slave functions of the UCA93 differ most from traditional Calibre AT and PCI bus based products. A user must make a pre-determined decision whether to put the adapter into master, slave transmitter or slave receiver modes. The adapter's own address is established as part of the Setup function. If the adapter is in master mode (the default) , it will not respond to its own address if this is generated by another master on the system. If the adapter is in either slave mode it will recognise its own address and an acknowledge will be generated on the I2C bus.

Slave Transmitter Function

When the BlockSlaveTransmittter function is called the user sends a data block of selectable size (max 2048bytes) into a buffer together a timeout time in seconds. When a master generates the adapter's own address (either write or read) the adapter will generate an acknowledge on the I2C bus.

If the address was even (master write) the adapter will assume that the master wishes to send a pointer to a location in the data block. The pointer can be one or two bytes. If the pointer is single the upper byte will default to 0x00. If the master fails to send a pointer both bytes will default to 0x00 and if the master send more than two bytes on the last two will be used – the others will be discarded. Data in the block will then be discarded so that the first byte aligns with the pointer sent. Note that the adapter will hold SCL low during this alignment which slows down the I2C bus – the master should be able to tolerate this.

If the address was odd (master read) the adapter will assume that the master does not wish to send a pointer and that data transmission should start from the beginning of the data block.

If the master has sent a pointer the master should then generate a restart followed by the adapter's read address which will be acknowledged.

Data is then transmitted from the adapter to the master starting at the pointer location. Remembering that the master is in charge of the length of the transfer and the generation of the stop, the adapter will transmit data to the master for as long as required. If the transfer is longer than the available data the last available byte in the block will be repeated; if it is shorter then not all the data in the block will be used.

When the transmission is terminated by the master the adapter will send a single byte back to the host.

The BlockSlaveTransmitterStatus function waits for this byte and the byte is interpreted as described the function definition. Note that this byte may return either as a result of a timeout (when the master fails to initiate a transfer to the adapter within the timeout seconds) or the master correctly terminating the transfer.

When the BlockSlaveTransmittter function terminates the adapter goes back into an idle state – it will not acknowledge its own address or repeat any other slave behaviour until the slave function is called again.

Users will note that the BlockSlaveTransmittter function is designed as much as possible to emulate the behaviour of an EEPROM. The only differences are that the block size is limited to 2048 byte, there may be some delay on the bus white data is aligned with the pointer and the function only runs once without being recalled. The function is also designed to keep the transmission going even if a mistake is made in the master's protocol although it is important that the master generates a stop eventually.

Slave Receiver Function

The BlockSlaveReceiver function is simpler than the previous function! When it is called the user must instruct the adapter on how many bytes it wishes to receive back (max 2048) and on a timeout in seconds.

When the master generates the adapter's own address the adapter will generate an acknowledge on the I2C bus. Since the adapter is about to receive data this should be an even (write) address.

The master then transmits data to the adapter and the adapter will acknowledge each byte then put it into the pre-determined size buffer to be transmitted to the host. The adapter will accept data from the master for as long as the master sustains the transfer. If too much data is sent by the master, the adapter keeps on acknowledging the master but stops putting data into the buffer. Likewise, if the master terminates the transfer before the buffer is full, the adapter pads out the buffer with 0xFFs to the predetermined size.

When the master terminates the transfer, the adapter completes sending the data block to the host together with a single status byte indicating how the transfer went, and the data is put into a global variable. The BlockSlaveReceiverStatus function waits for this status byte and the byte can be interpreted as in the function definition.

When the BlockSlaveReceiver function terminates the adapter goes back into an idle state – it will not acknowledge its own address or repeat any other slave behaviour until the slave function is called again.

If the master erroneously sends a read address when the BlockSlaveReceiver function has been called the adapter will send a fixed sequence of 0x55s until the master terminates the transfer.

When the THE MOST COMMONLY ASKED I2C QUESTIONS

General Questions

Question **I get corrupted transfers why is this?**
Answer The most likely reason for corrupted transfers is either incorrect bus termination or excessive capacitance - see the manual for details.

Question **Do you have software to talk to my.....?**
Answer Unfortunately there are too many I2C devices for us to be able to offer complete solutions - although we can supply a windows based application called WINI2C which is designed for those just starting I2C or wishing to perform simple I2C tasks, please contact our sales team or look on our web site, www.calibreuk.com for further information.

Question **I am trying to read from a device, the first time my software works fine but when I try again I can't get anything what's wrong?**
Answer Please check that you are changing the value of Setnack in accordance with the manual, it is likely that you have not made Setnack 1 for the last byte being read.

Windows Questions

Question **I have read the manual and still cannot get the communications to run. What do I do next?**
Answer Check that you have fully implemented the protocol between the Adapter and the other I2C devices see the device manufacturers data sheet for details.

Check that the software you have written is logically and syntactically correct - this is probably the most common cause of software faults we have to deal with.

Send us the following details:-

1)The link settings of the Adapter.
2)A sketch of the relevant I2C hardware including the location of bus termination.
3)The type and speed of processor within your PC and which operating system, you are running.
4)Brief software listings, or which Calibre software you are running.
5)The serial number of your I2C Adapter, or when you purchased it.

PLEASE EMAIL YOUR QUERY TO: techsupport@calibreuk.com
OR FAX YOUR QUERY TO: 44-1274-730960

We will endeavour to help you.